Udacity Machine Learning Engineer Nanodegree Capstone project report: Dog Breed Classifier

Vasileios-Marios Gkortsas

Contents

1 Definition

1.1 Project Overview

The goal of the project is to learn how to build a pipeline to process real-world, user-supplied images. Given an image of a dog, the algorithm will identify an estimate of the canine's breed. If supplied an image of a human, the code will identify the resembling dog breed. The Machine Learning architecture we will use is the Convolution Neural Networks (CNN) one and we will follow 2 training approaches. In the first one, we will create and train the CNN from scratch. In the second one, we will apply transfer learning where we will use as a base model a well-known sophisticated architecture (ResNet50, VGG16) together with the corresponding training weights and we will modify and train only the last layer in order to adjust the architecture to our needs.

1.2 Problem statement

Classification of an image's content is the most common Computer Vision task. We will build different versions of image's classification. The first version will be a crude classifier, where the workflow will detect whether the input image has a human face or a dog. Then, using CNNs we can move to more advanced image classifiers, which will tell us the dog's breed if the image has a dog, or which dog breed the human looks like, if the image has a human face.

We will use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. But OpenCV does not include a dog classifier, so we will have to use CNNs for dog breed classification. We will create 2 versions of CNNs for dog breed classification. The first one will be a CNN architecture from scratch and the second one will be a pre-trained sophisticated architecture in which we will modify and train only the last layer in order to adjust the predicted number of classes to the number of dog breeds. The second approach is called transfer learning.

1.3 Metrics

The CrossEntropyLoss function provided by Pytorch will be used to evaluate the train and validation loss of the CNN architecture. The accuracy, defined as the number of times a predicted dog breed label is equal to the reference dog breed label divided by the total number of testing dog images, can be used to estimate the test accuracy of the CNN architecture.

2 Analysis

2.1 Data Exploration

Both the dog and human faces datasets are provided by Udacity and can be either downloaded in the local machine or used directly from the Udacity workspace. We will mainly focus on the dog dataset, since we will train the CNN architecture on this one. The dog dataset folder consists of 3 subfolders named train, valid and test, which will be used in each of the corresponding phases of the workflow. Each of these 3 subfolders has 133 subfolders, where each one of them corresponds to the 133 dog breeds. Figure 1

shows an example of the breed subfolders inside the train sub-folder of the dog dataset. Each of the breed sub-folders has images of the dogs.

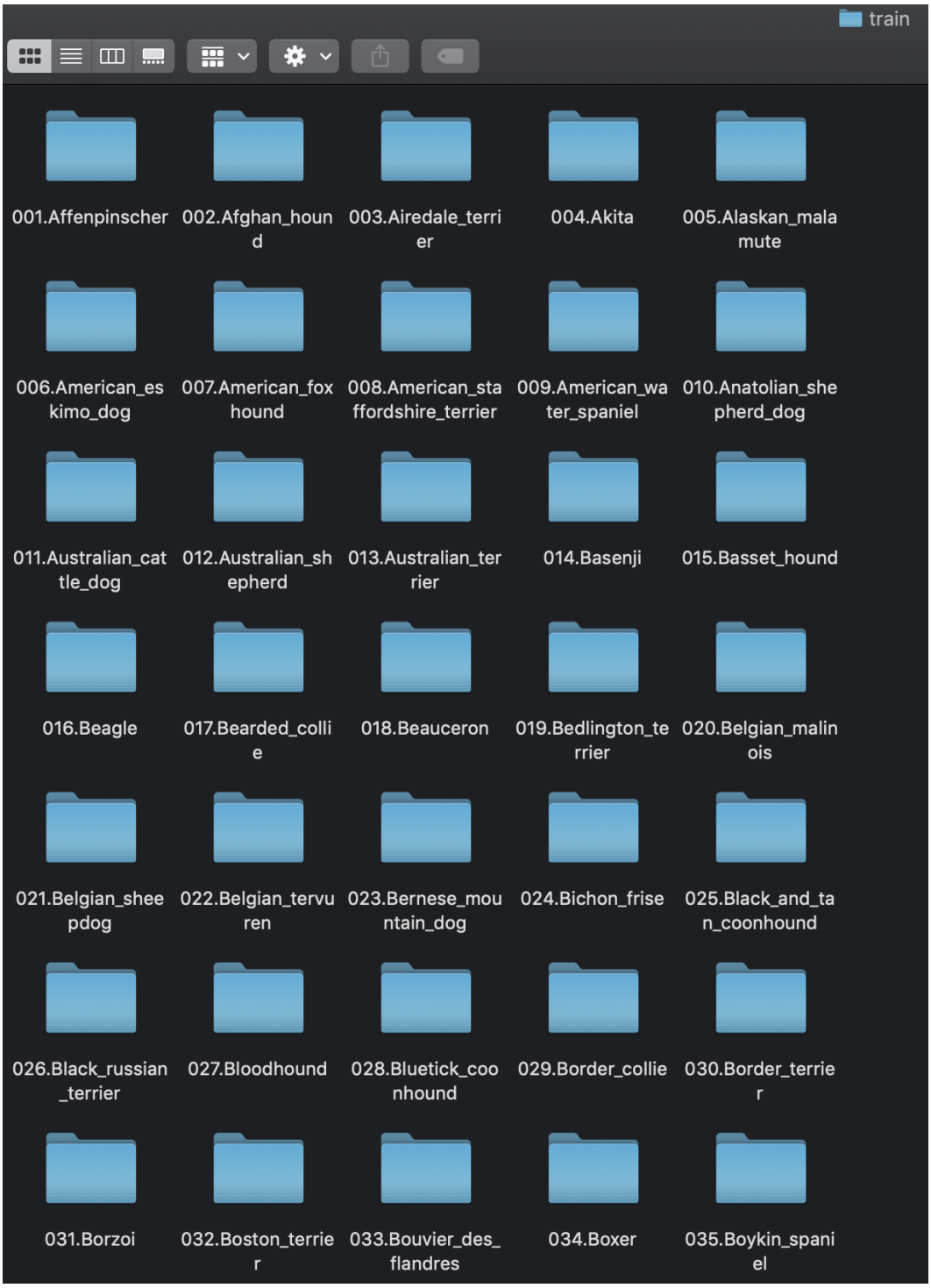

Fig. 1. Folders that contain images of different breeds of the dog dataset.

Figure 2 shows the images of a sub-folder of the train dataset named "Alaskan malamute".

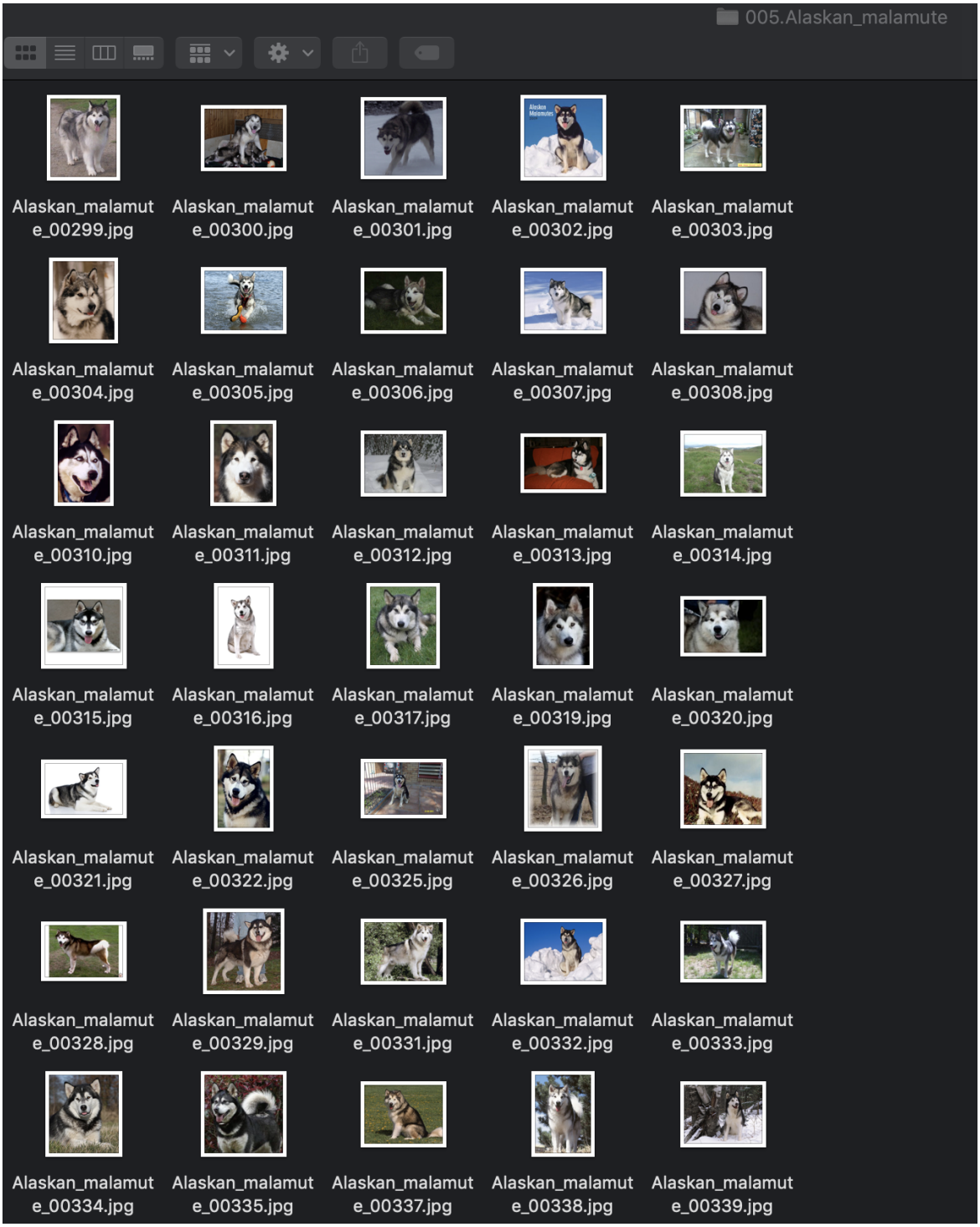

Fig. 2. Images of "Alaskan malamute" used for training.

The inputs to the CNN will come from these folders after they are converted to Pytorch tensors.

2.2 Algorithms and Techniques

First, in order to roughly classify whether a human face is included in the image, we use OpenCV's implementation of Haar feature-based cascade classifiers. OpenCV provides many pre-trained face detectors, stored as XML files and we chose to use the "haarcascade

frontalface alt". We extract the pre-trained face detector "haarcascade frontalface alt", we load a color (BGR) image, we convert the image to grayscale, we find the faces in the image and we can also get the bounding box for each detected face.

In order to roughly detect whether a dog is included in the image or not, we use a pre-trained sophisticated architecture like ResNet50 or VGG16 which have been trained on an extensive set of images named ImageNet, which contains more than 14 million images, which belong to 1000 different classes. Within these 1000 classes there are 118 different dog breeds. Thus, using the pre-trained architectures, we extract the predicted class index and if it belongs within the range of class indices which correspond to dog breeds in ImageNet i.e. between 151 to 268, the detector returns True.

However, the number of dog breeds, that the pre-trained architectures have been trained on in the ImageNet dataset, is 118, while the number of dog breeds in our dataset is 133. Thus, our data have more classes and thus we either need to design our own CNN architecture or modify the pre-trained ones.

Thus, I designed the following CNN architecture from scratch. Figure 3 shows the architecture I designed from scratch and Figure 4 shows the number of training parameters for each layer.

My proposed CNN architecture from scratch consists of 5 Conv2D layers, with each one of them being followed by a BatchNorm2d layer, a ReLU activation function and a MaxPooling2D layer. The filter size is 5 by 5 for the first 2 layers and 3 by 3 for the last 3 since the image size (height, width) decreases and we also want to extract higher level features. On the other hand, the number of filters increases, as we go deeper in the network. I added a batch normalization layer after each convolutional layer to allow each convolutional layer to learn by itself a bit more independently of other layers and thus speeding up the learning of the whole network. The MaxPooling2D layers were used for multiscale feature extraction. After, I used a global average pooling operation, which is applied for each channel independently, to get the average of the features for each channel and gives as output 1D tensor. I then add a fully connected layer with ReLU activation function followed by a dropout with dropout probability 50% to prevent overfitting and then a final fully connected layer, where the number of neurons is equal to the number of dog breeds. This final layer will provide the probability scores that an image belongs to each of the 133 breeds. I did not add a softmax activation function and the output of the layer is a logit since the loss function (i.e. CrossEntropyLoss()) in Pytorch takes care of that.

```python
# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5)
        self.bn1 = nn.BatchNorm2d(num_features=16)

        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5)
        self.bn2 = nn.BatchNorm2d(num_features=32)

        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3)
        self.bn3 = nn.BatchNorm2d(num_features=64)

        self.conv4 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3)
        self.bn4 = nn.BatchNorm2d(num_features=128)

        self.conv5 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3)
        self.bn5 = nn.BatchNorm2d(num_features=256)

        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.drop = nn.Dropout(0.5)

        self.fc1 = nn.Linear(256, 512)
        self.fc2 = nn.Linear(512, len(train_data.classes))


    def forward(self, x):

        x = F.relu(self.bn1(self.conv1(x)))
        x = self.maxpool(x)
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.maxpool(x)
        x = F.relu(self.bn3(self.conv3(x)))
        x = self.maxpool(x)
        x = F.relu(self.bn4(self.conv4(x)))
        x = self.maxpool(x)
        x = F.relu(self.bn5(self.conv5(x)))
        x = self.maxpool(x)

        x = F.adaptive_avg_pool2d(x, (1, 1))
        x = x.view(x.shape[:2])

        x = F.relu(self.fc1(x))
        x = self.drop(x)
        x = self.fc2(x)

        return x

#-#-# You so NOT have to modify the code below this line. #-#-#

# instantiate the CNN
model_scratch = Net()
# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
```

Fig. 3. My proposed CNN architecture from scratch.

```
conv1.weight      shape: torch.Size([16, 3, 5, 5])       Parameters: 1200
conv1.bias        shape: torch.Size([16])        Parameters: 16
bn1.weight        shape: torch.Size([16])        Parameters: 16
bn1.bias          shape: torch.Size([16])        Parameters: 16
conv2.weight      shape: torch.Size([32, 16, 5, 5])      Parameters: 12800
conv2.bias        shape: torch.Size([32])        Parameters: 32
bn2.weight        shape: torch.Size([32])        Parameters: 32
bn2.bias          shape: torch.Size([32])        Parameters: 32
conv3.weight      shape: torch.Size([64, 32, 3, 3])      Parameters: 18432
conv3.bias        shape: torch.Size([64])        Parameters: 64
bn3.weight        shape: torch.Size([64])        Parameters: 64
bn3.bias          shape: torch.Size([64])        Parameters: 64
conv4.weight      shape: torch.Size([128, 64, 3, 3])     Parameters: 73728
conv4.bias        shape: torch.Size([128])       Parameters: 128
bn4.weight        shape: torch.Size([128])       Parameters: 128
bn4.bias          shape: torch.Size([128])       Parameters: 128
conv5.weight      shape: torch.Size([256, 128, 3, 3])    Parameters: 294912
conv5.bias        shape: torch.Size([256])       Parameters: 256
bn5.weight        shape: torch.Size([256])       Parameters: 256
bn5.bias          shape: torch.Size([256])       Parameters: 256
fc1.weight        shape: torch.Size([512, 256])  Parameters: 131072
fc1.bias          shape: torch.Size([512])       Parameters: 512
fc2.weight        shape: torch.Size([133, 512])  Parameters: 68096
fc2.bias          shape: torch.Size([133])       Parameters: 133
```

Fig. 4. Number of training parameters for each layer.

After training for 20 epochs, the testing accuracy is 28% (i.e. 238 testing images out of the 836 total ones were classified with a predicted breed equal to the reference dog breed). The expected goal of testing accuracy was 10% and the proposed architecture did better.

I want to further improve the accuracy of the testing dataset and we will use transfer learning for that. In transfer learning, we use a well-known sophisticated architecture together with the corresponding training weights and we will modify and train only the last layer in order to adjust the architecture to our needs.

I used ResNet50 as the model architecture for transfer learning. I froze all layers apart from the last fully connected one where I changed the number of output neurons, which give the probability of the class, from 1000 neurons (since ResNet50 is trained on the ImageNet dataset where the images belong to 1000 classes) to 133 so we fit our needs and trained for 20 epochs.

ResNet50 is trained on more than 14 million images which belong to 1000 different classes. This means that the first layers i.e. convolutional layers, are trained to extract features from different types of images. Our dog dataset consists of a much smaller number of images, i.e. 6680 images belonging to 133 distinct classes, and thus is more specific, so we need to fine-tune the final layer by changing the number of output neurons (i.e. 1000) to be equal to the number of dog breeds. This is the core idea of transfer learning.

I repeated the procedure by applying transfer learning using the VGG16 architecture, and the results were almost the same. The test accuracy when applying transfer learning using the ResNet50 was 87% (732/836), while when using the VGG16 architecture was 86% (723/836).

With transfer learning we achieve much better testing accuracy (87%) than the CNN architecture from scratch (28%) and better than the 60% accuracy that was the project's requirement.

```python
import os
from torchvision import datasets

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

train_data_path = "/data/dog_images/train"
valid_data_path = "/data/dog_images/valid"
test_data_path = "/data/dog_images/test"

transform_img = transforms.Compose([
            transforms.Resize(256),     # Resize the image to 256×256 pixels
            transforms.CenterCrop(224), # Crop the image to 224×224 pixels about the center
            transforms.ToTensor(),      # Convert the image to PyTorch Tensor data type
            transforms.Normalize(       # Normalize the image by setting its mean and standard deviation to the specifi
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225])])

transform_label = transforms.Compose([transforms.ToTensor()])

BATCH_SIZE = 64

train_data = datasets.ImageFolder(root=train_data_path, transform=transform_img)
train_data.samples = [(d, torch.tensor(s)) for d, s in train_data.samples]
train_data_loader = torch.utils.data.DataLoader(train_data, batch_size=BATCH_SIZE, shuffle=True, num_workers = 0)

valid_data = datasets.ImageFolder(root=valid_data_path, transform=transform_img)
valid_data.samples = [(d, torch.tensor(s)) for d, s in valid_data.samples]
valid_data_loader = torch.utils.data.DataLoader(valid_data, batch_size=BATCH_SIZE, shuffle=True, num_workers = 0)

test_data = datasets.ImageFolder(root=test_data_path, transform=transform_img)
test_data.samples = [(d, torch.tensor(s)) for d, s in test_data.samples]
test_data_loader = torch.utils.data.DataLoader(test_data, batch_size=BATCH_SIZE, shuffle=True, num_workers = 0)

loaders_scratch = {}
loaders_scratch['train'] = train_data_loader
loaders_scratch['valid'] = valid_data_loader
loaders_scratch['test'] = test_data_loader
```

Fig. 5. Build datasets and PyTorch dataloaders.

3 Methodology

3.1 Data Preprocessing

In RGB, a color is defined as a mixture of pure red, green, and blue lights of various strengths. Each of the red, green and blue light levels is encoded as a number in the range 0...255, with 0 meaning zero light and 255 meaning maximum light. Since the images can have different shapes, I first resize them to be 256 by 256 pixels. The pre-trained CNN models expect batches of images of dimensions $3 \times 224 \times 224$ and for this reason I cropped the image to 224×224 pixels about the center. Since PyTorch models can only accept torch.tensor data type as inputs, all the pre-processed images need to be converted to tensors. Finally, I normalize the image by setting its mean and standard deviation to specified values, i.e. mean=$[0.485, 0.456, 0.406]$ and std=$[0.229, 0.224, 0.225]$. The conversion of the labels to tensors does not happen within the image transform, but separately.

Finally, in order to load batches of images and their corresponding labels from the PyTorch datasets I build dataloaders (Figure 5), which are used as iterators in the training process.

3.2 Implementation and Refinement

Figure 3 shows the implementation of the proposed CNN architecture in PyTorch from scratch. Since Udacity gives us access to GPUs, i moved the PyTorch tensors to GPU to speed up the training. The loss function I used is the CrossEntropyLoss from PyTorch,

since it is a multiclass classification problem. The optimizer to update the weights is the Adam optimizer (Figure 6).

```
### TODO: select loss function
criterion_scratch = nn.CrossEntropyLoss()

### TODO: select optimizer
lr = 1e-3
optimizer_scratch = optim.Adam(model_scratch.parameters(), lr)
```

Fig. 6. Loss function and optimizer.

The training process is designed as follows: for each epoch, I load batches of training and validation data from the corresponding dataloaders, move them to the GPU (if available), clear all optimized variables (PyTorch standard), calculate the model output, calculate the loss, perform backward propagation and update the model parameters by one step. If the validation loss has decreased, I save the model. The train function returns the model as its output.

The test function is designed in the same way: I load batches of testing data from the corresponding dataloaders, predict the model output and find the index that has the largest value in the output vector which consists of 133 elements, equal to the number of dog breeds. The index which corresponds to the element with the maximum value of the output tensor gives us the predicted dog breed.

I also use transfer learning to improve the accuracy, choosing ResNet50 as the model architecture for transfer learning. I froze the weights of all layers so they are not re-trained, apart from the ones of the last fully connected one. In the last fully connected layer, I changed the number of output neurons, which give the probability of the class, from 1000 neurons (since ResNet50 is trained on the ImageNet dataset where the images belong to 1000 classes) to 133 so it fits the problem's needs and trained the last layer for 20 epochs. In addition, I applied transfer learning using the VGG16 architecture as the base model.

```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
model_transfer = models.resnet50(pretrained=True)
# Freeze model weights
for param in model_transfer.parameters():
    param.requires_grad = False
```

```
model_transfer.fc = torch.nn.Linear(model_transfer.fc.in_features, len(train_data.classes))

if use_cuda:
    model_transfer = model_transfer.cuda()
```

Fig. 7. Transfer learning.

4 Results

4.1 Model Evaluation, Validation and Justification

After training the model (CNN from scratch, applying transfer learning and modifying the final fully connected layer with base model being Reset50 and VGG16), I test the trained models on the testing set. The CNN from scratch gave testing accuracy of 28% (i.e. 238 testing images out of the 836 total ones were classified with a predicted breed equal to the reference dog breed). The transfer learning model using ResNet50 as base model gives accuracy of 87% (732/836), while the transfer learning model using VGG16 as base model gives accuracy of 86% (723/836). All accuracies are better than the minimum required ones, i.e. 10% for the model from scratch and 60% for the transfer learning one.

ResNet50 is trained on more than 14 million images which belong to 1000 different classes. This means that the first layers i.e. convolutional layers, are trained to extract features from different types of images, including dog ones. Such a feature extractor is much better than the one in the model form scratch, which extracts features from 6680 dog images.

Finally, I combine all the different functions I wrote in the Jupyter notebook in order to create an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, if a dog is detected in the image it returns the predicted breed, if a human is detected in the image it returns the resembling dog breed and if neither is detected in the image it indicates an error.

First, the pre-trained ResNet50 model without transfer learning is used to tell whether the input image contains a dog and if it does the ResNet50 model after transfer learning gives the predicted breed of the dog in the image. If the pre-trained ResNet50 model without transfer learning determines that there is no do in the image, then the OpenCV face detector examines whether there is a human face in the image and if there is the ResNet50 model after transfer learning provides an estimate of the dog breed that the human image is resembling the most. If neither dog, nor human are detected in the image a message is printed saying that.

Figure 8 shows the function "*run_app*" which does what i described above.

```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

ResNet50 = models.resnet50(pretrained=True)

def run_app(img_path):
    ## handle cases for a human face, dog, and neither

    img = cv2.imread(img_path)
    # convert BGR image to RGB for plotting
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    # display the image, along with bounding box
    plt.imshow(cv_rgb)
    plt.show()

    if dog_detector_model(ResNet50, img_path)==True:
        print('hello dog!')
        print('Your breed is ...')
        print(predict_breed_transfer(img_path))
    elif face_detector(img_path)==True:
        print('hello human!')
        print('Your look like a ...')
        print(predict_breed_transfer(img_path))
    else:
        print('picture has neither dog nor human')
```

Fig. 8. Function "*run_app*".

Figure 9 shows the supporting functions of the function "*run_app*".

```python
def model_predict(model, img_path):
    '''
    Use pre-trained model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to  model's prediction
    '''

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image

    transform = transforms.Compose([     # defining a variable transform which is a combination
            transforms.Resize(256),      # Resize the image to 256×256 pixels
            transforms.CenterCrop(224),  # Crop the image to 224×224 pixels about the center
            transforms.ToTensor(),       # Convert the image to PyTorch Tensor data type
            transforms.Normalize(        # Normalize the image by setting its mean and standard
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225])])

    img = Image.open(img_path)
    img_t = transform(img) # shape: (3,224,224)
    batch_t = torch.unsqueeze(img_t, 0) # shape: (1,3,224,224)
    #batch_t = batch_t.cuda()
    # put the model in eval mode
    model.eval()
    out = model(batch_t) # shape: (1,1000)

    return torch.argmax(out, dim=1).item() # predicted class index
```

```python
def dog_detector_model(model, img_path):
    ## TODO: Complete the function.
    ## TODO: Complete the function.
    prediction = model_predict(model, img_path)
    return ((prediction <= 268) & (prediction >= 151)) # true/false
```

```python
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:].replace("_", " ") for item in loaders_transfer['train'].dataset.classes]

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    img = Image.open(img_path)
    img_t = transform_img(img) # shape: (3,224,224)
    batch_t = torch.unsqueeze(img_t, 0) # shape: (1,3,224,224)
    batch_t = batch_t.cuda()

    model_transfer.eval()
    out = model_transfer(batch_t) # shape: (1,133)
    index = torch.argmax(out, dim=1).item()


    return class_names[index]
```

Fig. 9. Supporting functions of function "*run_app*".

I applied the "*run_app*" function on images that I uploaded. Out of the 6 dog images, it correctly identified that there is a dog in all of them and it correctly predicted the dog's breed in 5 of them. I uploaded 2 images, one of a cat and one of a rabbit and it correctly identified that they include neither a human nor a dog. The fun part of the project is the part that when a human is identified, the app tries to figure out which dog breed it resembles the most.

Figure 10 shows examples of applying the "*run_app*" on various images.

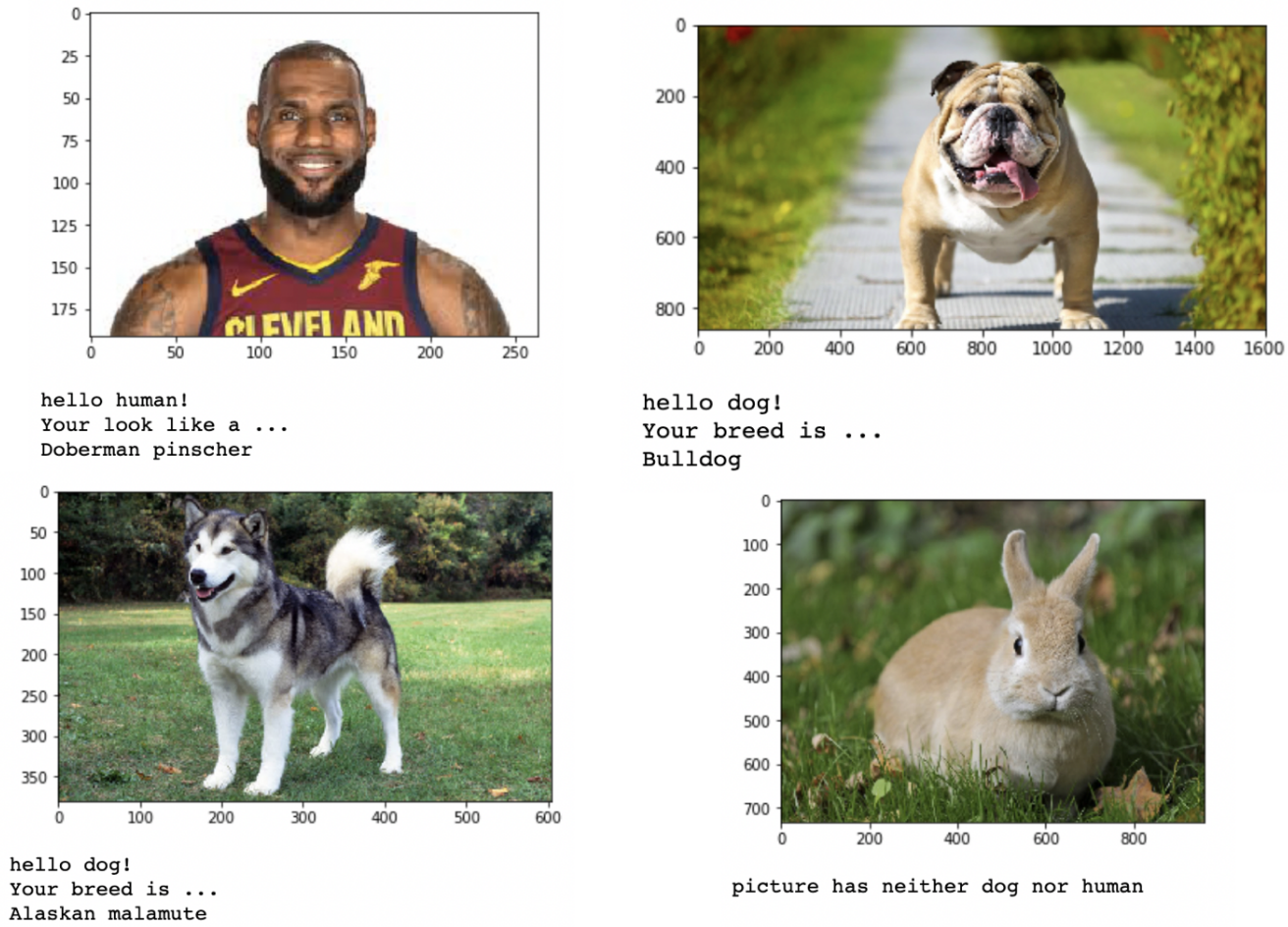

Fig. 10. Examples of applying the function "*run_app*" on various images.

5 Conclusion

In this project I developed a classification pipeline which first identifies whether a given input image contains a human face or a dog. Then, if it contains a dog it classifies it to a dog breed and if it contains a human face it finds the dog breed the human face resembles the most.

OpenCV can do human face detection. But dog breed classification happens through a CNN architecture. I used 2 types of CNN architectures. One developed from scratch and one that uses transfer learning where I use as base model a pre-trained sophisticated one, like the ResNet50, and modify and train only its last fully connected layer so the number of output neurons matches the number of dog breeds. The transfer learning model provides much better accuracy than the one from scratch (87% vs 28%) and is the one that I choose to use when I give my images as inputs.

Out of the 6 dog images, it correctly identifies that there is a dog in all of them and it correctly predicts the dog's breed in 5 of them. When I upload an image of a human face it figures out which dog breed it resembles the most. Thus, the algorithm performs better than I expected.

5.1 Future work - Improvements

Possible improvements of the CNN models, in order to get better accuracy, could be the following:

- Use a different and better architecture as base model for transfer learning.
- Train for higher number of epochs and deal with learning rate in a more sophisticated way in order to get better prediction accuracy.
- Perform data augmentation to increase the size of the training set.